AD-772 509

RECURSIVE DATA STRUCTURES

C. A. R. Hoare

Stanford University

Prepared for:

Advanced Research Projects Agency
National Science Foundation

October 1973

STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-223

STAN-CS-73-400

RECURSIVE DATA STRUCTURES

BY

C. A. R. HOARE

SUPPORTED BY

ADVANCED RESEARCH PROJECTS AGENCY
ARPA ORDER NO. 2494
PROJECT CODE 3D30

OCTOBER 73

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

36

Recursive Data Structures

C. A. R. Hoare

Abstract. The power and convenience of a programming language may be enhanced for certain applications by permitting data structures to be defined by recursion. This paper suggests a pleasing notation by which such structures can be declared and processed; it gives the axioms which specify their properties, and suggests an efficient implementation method. It shows how a recursive data structure may be used to represent another data type, for example, a set. It then discusses two ways in which significant gains in efficiency can be made by selective updating of structures, and gives the relevant proof rules and hints for implementation. It is shown by examples that a certain range of applications can be efficiently programmed, without introducing the low-level concept of a reference into a high-level programming language.

The work on this paper was supported in part by National Science Foundation under grant number GJ 36473X and ARPA Research Contract DAHC 15-73-C-0435.

1. Introduction

In a language such as ALGOL 68 [1] or PL/I [2], a central role is played by the concept of a reference or POINTER. In ALGOL 68, the reference underlies the treatment of ordinary variables, result parameters, data structuring, dynamic storage allocation, indirect addressing, etc., and in PL/I they are also used for value parameters, and even for input/output. However there are many reasons to believe that the introduction of references into a high-level language is a seriously retrograde step:

(1) It reintroduces the same unpleasant confusion between addresses and their contents which afflicts machine code programmers.

(2) In ALGOL 68, confusion is doubly confounded by complex coercion and balancing rules.

(3) In PL/I the explicit allocation and deallocation of storage affords unbounded scope for complexity and error.

(4) The variables subject to change by a program statement are no longer manifest from the form of the statement. For example, if x and y are reference variables

```
x := y;
```

obviously changes x , but a statement in ALGOL 68 like

```
x := y+1;
```

may change a , or b , or any other variable of appropriate type: one variable it can't possibly change is x !

(5) It is possible to retain a reference value to an area of local workspace which has been deallocated. In PL/I this can cause disaster without warning; in ALGOL 68 certain rather complex rules ensure that the danger can sometimes (but not always) be averted by a compile-time check. This is known as the problem of the "dangling reference".

(6) In distinction from values of all normal types (integers, reals, arrays, strings, files,...) the value of a reference can never be input to a program, nor output from it (except possibly in a total post mortem dump).

(7) The use of references reduces the efficiency of execution on machines with instruction lookahead, data prefetch, pipelines, slave stores or paging systems, counteracting all these laudable attempts by hardware to make a machine seem faster or larger than it really is.

(8) When data is to be held permanently or temporarily on backing store (e.g. files on tape or disk), the use of references can create insuperable difficulties to implementor, user, or both.

(9) Proof methods for dealing with a language which permits general pointers are significantly more complicated, whether the pointers are used or not.

There appears to be a close analogy between references in data and jumps in a program. A jump is a very powerful multipurpose tool, present in the object code produced by compilers for almost every machine. But it is also an undisciplined feature, which can be used to create wide interfaces between parts of a program which appear to be disjoint. That is why a high level programming language like ALGOL 60 has introduced a range of program structures such as compound statements, conditional statements, while statements, procedure statements, and recursion to replace many of the uses of the machine code jump. Indeed perhaps the only remaining purpose of the jump is to indicate irreparable breakdown in the structure of the program. Similarly, if references have any rôle in data structuring it may be a purely destructive one. It would therefore seem highly desirable to attempt to classify all those special purposes to which references may be put, and to replace them in a high level language by more structured principles and notations. In this task, it is encouraging that ALGOL 60 [3] has already isolated two such uses, namely the procedure parameter and the variable length array, and has dealt with them without introducing the reference concept. Furthermore ALGOL W [4] and PASCAL [5] have introduced references as representations of many-one relationships in a relational network, and have done so in a manner which mitigates many of the disadvantages mentioned above (as compared with (say) ALGOL 68 or PL/I).

One of the main reasons for using stored machine addresses is that the amount of storage that will be required by an item of data is not known to the compiler. In this paper we will consider a class of data structures for which the amount of storage required can actually vary during the lifetime of the data; and we will show that it can be satisfactorily accommodated in a high level language using solely high level problem-oriented concepts, and without the introduction of references.

2. Concepts and Notations

The method of specifying the set of values of a data space by recursion has long been familiar to modern logicians. For example, the propositions treated in conventional propositional calculus may be defined by the following four rules:

1. All proposition letters are propositions.
2. If p is a proposition then so is ¬ p .
3. If p and q are propositions, then so are (p & q) and (p ∨ q) .
4. All propositions can be obtained from proposition letters by a finite number of applications of the above rules.

When the set of propositions as defined above is treated as an object of mathematical study, it is known as a "generalized arithmetic"; and an additional axiom is postulated:

5. Two propositions are equal only if they have been obtained by the same rule from equal components.

Exactly the same idea is familiar to programmers in the use of the BNF notation for the definition of programming language grammars. For example, propositions could be defined:

```
⟨proposition⟩ ::= ⟨proposition letter⟩|
                  ¬ ⟨proposition⟩|
                  (⟨proposition⟩ & ⟨proposition⟩)|
                  (⟨proposition⟩ ∨ ⟨proposition⟩)
⟨proposition letter⟩ ::= ⟨letter⟩
```

Both these methods of defining data not only specify the abstract structure of the data, they also state how any value can be represented as a linear stream of characters, for example:

(P & (¬ P ∨ Q)) .

However, we wish to abstract from the external appearance of the data, and concentrate on its structural properties. This abstraction is familiar to an algebraist, who calls the resulting data space a word algebra on a given finite set of generators. A generator is a function which maps its parameter(s) onto the larger structure of which they are immediate components. A generator with no parameters is known as a constant. In the case of propositions, four generators are required:

(1) prop: letter → proposition;
which converts any letter into a proposition letter (logicians often use a different type font for this).

(2) neg: proposition → proposition;
which constructs the negation of its argument.

(3) conj, disj: proposition x proposition → proposition;
which takes two arguments and whose result is their conjunction or disjunction respectively.

In symbolic manipulation programs, it is common to deal with variables, parameters, and functions whose values range over data spaces such as logical propositions. In a language like PASCAL, which permits and encourages the programmer to define and use his own data types, it seems reasonable to permit him to use recursive definitions when necessary. A possible notation for such a type definition was suggested by Knuth [6]; it is a mixture of BNF (the | symbol) and the PASCAL definition of a type by enumeration:

```
type proposition = (prop (letter) | neg (proposition) |
                    conj, disj (proposition, proposition));
```

It is assumed that the type "letter" has been predefined, for example as a subrange of characters

```
type letter = 'A' .. 'Z'   .
```

The effect of this type definition is threefold:

(1) it introduces the name of the type;
(2) it introduces the names of its generators;
(3) it gives the number and types of the argument(s) of the generators (if any).

Type definitions of this sort were suggested by McCarthy in [7].

The type is intended to be used to declare variables, parameters (and functions) ranging over the type, e.g.:

```
P1, P2: proposition;
```

and the generators can be used to define values of the type, e.g., the sequence of instructions:

```
P1 := prop ('P');
P2 := neg (P1);
P2 := disj (P2, prop ('Q'));
P2 := conj (P1,P2);
```

would leave as the value of P2 a proposition which would normally be written:

$$(P \,\&\, (\neg P \vee Q)) \quad .$$

In most languages with references, the use of recursive type definitions is permitted only if the recursive components of each structure are declared as references. This seems to be a rather low level machine oriented restriction; after all, we do not insist that recursive calls of a procedure should be signalled by such special notations. It is true that a recursive data structure which is held in a conventionally addressed main store will usually be represented by references, but it seems a good idea that the programmer should be encouraged to ignore the machine-oriented details of the representation (just as he ignores details of the implementation of recursive procedures), and should concentrate on the more pleasant abstract properties of the structure. The implementor should also have the freedom to use a different representation, for example, when the data is held on a backing store. Thus the programmer may, if he wishes, imagine a machine which allocates a fixed amount of space to hold the current value of a variable of recursive type; and if it is called upon to fit in a larger value, it adopts the same expedient that we do -- it merely writes smaller!

In defining operations on a data structure, it is usually necessary to enquire which of the various forms the structure takes, and what are its components. For this, I suggest an elegant notation which has been implemented by Fred McBride in his pattern-matching LISP [8]. Consider for example a function intended to count the number of &s contained in a proposition. Like many functions operating on recursively defined data, it will be recursive:

```
(1)      function andcount (p: proposition): integer;
(2)         andcount := cases p of
(3)                     (prop(c) → 0|
(4)                      neg(q) → andcount(q)|
(5)                      conj(q,r) → andcount(q) + andcount(r)+1|
(6)                      disj(q,r) → andcount(q) + andcount(r));
```

Line (1) declares andcount to be an integer-valued function of one proposition, known as p in the body of the function.

Line (2) states that the result of andcount is assigned by computing the following expression. This is a "case expression" whose effect will depend on the value of p .

Line (3) states that if the value of p is a proposition letter c , the result is zero.

Line (4) states that if the value of p is a negation, let q be the negated proposition and the result is found by computing the andcount of q .

Line (5) states that if the value of p is a conjunction, let q and r be the names of its components, and the result is one more than the sum of the ancounts for q and r .

Note that the identifiers c , q , r are like formal parameters: they are declared by appearing in the parameter list to the left of the arrow, and their scope is confined to the right hand side of the arrow, only as far as the vertical bar. Their types are determined by the types given in the declaration of the corresponding generator, e.g., c is a letter, and q and r are propositions. We shall insist, for the time being, that the programmer shall not make assignments to these variables.

The language feature described above is evidently capable of expressing all the functional aspects of LISP, and many of the procedural aspects as well. For example, the list structure of LISP can be defined:

```
type list = (unit (identifier) | cons (list, list))
```

where the type identifier is assumed to be predefined. The function cons is defined as part of this declaration. The other LISP basic functions can be programmed:

```
function car (ℓ: list): list;
   car := cases ℓ of (atom (id) → error|
                      cons (left, right) → left);
function cdr (ℓ: list): list; ... similar ...
function atom (ℓ: list): Boolean;
   atom := cases ℓ of (unit (any) → true | cons (x,y) → false);
function equals (ℓ1,ℓ2): Boolean;
equals := cases ℓ1 of
           (unit (id1) → cases ℓ2 of (unit (id2) → id1 = id2|
                                      cons (x,y) → false)|
            cons (x1,y1) → cases ℓ2 of (unit (id2) → false|
                                        cons (x2,y2) →
                              equals (x1,y1) & equals (x2,y2)));
```

In practice, the cases notation will often be found more convenient, clear, and less prone to error than the functions car , cdr , and atom . For example, the familiar append function may be written:

```
function append(ℓ1,ℓ2: list): list;
append := cases ℓ1 of
          (unit (id) → if id = NIL then ℓ2 else error|
           cons (first, rest) → cons (first, append (rest, ℓ2)));
```

Just as LISP can be embedded in any language which permits recursive data structures, so can all recursive data structures be represented as LISP lists, and processed by LISP functions. For example

```
conj ('P',disj(neg('P'),'Q'))
```

can be represented (in S-expression form):

```
('CONJ 'P ('DISJ ('NEG 'P) 'Q))
```

An andcount function for propositions represented in this way would be:

```
andcount := (atom(ℓ) → 0,
                car(ℓ) = 'NEG → andcount(cadr(ℓ)),
                car(ℓ) = 'CONJ → andcount(cadr(ℓ)) + andcount(caddr(ℓ)) + 1,
                car(ℓ) = 'DISJ → andcount(cadr(ℓ)) + andcount(caddr(ℓ)));
```

Note the arrows in this program are LISP conditionals. This example illustrates some of the advantages of the type declaration for recursive data structures:

(1) The check against the error of applying the function to a structure which is *not* a proposition can be made more rigorous, and can occur at compile time rather than run time.

(2) It is easier to check that all cases have been dealt with.

(3) The formal parameters seem to be more readable and perspicuous than the abbreviations car , cadr , caddr , etc.

In the next section it will be shown how a compiler can sometimes take advantage of the extra information supplied by a type declaration to secure more compact representations and more efficient code than is usually achieved in LISP.

To summarize the notational conventions introduced in this section, here are the syntax specifications of recursive type declarations and case expressions:

```
⟨type declaration⟩ ::= type ⟨type identifier⟩ = (⟨generator list⟩)
⟨generator list⟩ ::= ⟨generator⟩|⟨generator⟩⟨or symbol⟩⟨generator list⟩
⟨or symbol⟩ ::= |                          (i.e., vertical stroke)
⟨generator⟩ ::= ⟨generator identifier⟩|⟨generator identifier⟩(⟨type list⟩)
⟨type list⟩ ::= ⟨type⟩|⟨type⟩,⟨type list⟩
⟨case expression⟩ ::= cases ⟨expression⟩ of (⟨case list⟩)
⟨case list⟩ ::= ⟨case clause⟩|⟨case clause⟩⟨or symbol⟩⟨case list⟩
⟨case clause⟩ ::= ⟨pattern⟩ → ⟨expression⟩
⟨pattern⟩ ::= ⟨generator identifier⟩(⟨formal parameter list⟩)|
              ⟨generator identifier⟩
⟨formal parameter list⟩ ::= ⟨formal parameter⟩|
                            ⟨formal parameter⟩,⟨formal parameter list⟩
⟨formal parameter⟩ ::= ⟨identifier⟩
```

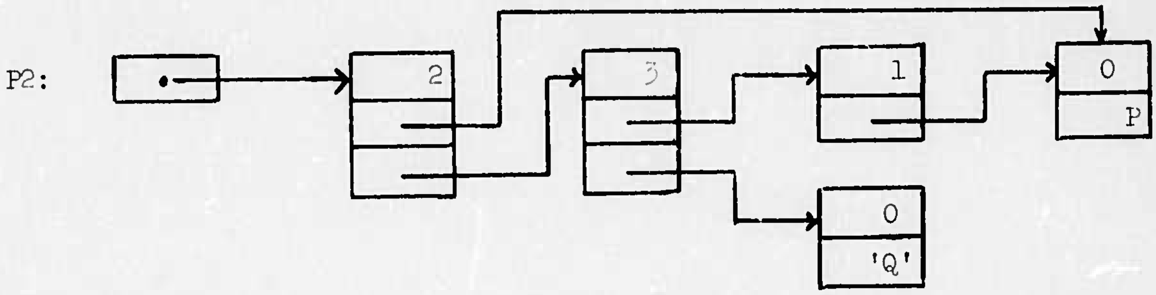
3. Implementation

The normal method of representing a recursive data structure for processing in the main store of a computer is as a tree using machine addresses to link the nodes, and a small integer, called a tag, in each node (or with the address) to indicate which of the generators was used to define this node. Each node contains as components the values of the arguments of the generator, which may be themselves addresses of other nodes, or many be just simple values.

For example, in the case of a proposition, the name of the generator is represented by an integer between 0 and 3 . If the node is a proposition letter (tag 0), this will be followed immediately by a representation of the letter. If it is a negation, the tag 1 is followed by the address of the negated proposition. In the remaining two cases, the code 2 or 3 is followed by a pair of locations, pointing to the components of the conjunction or disjunction. Thus the value

$$(P \& (\neg P \vee Q))$$

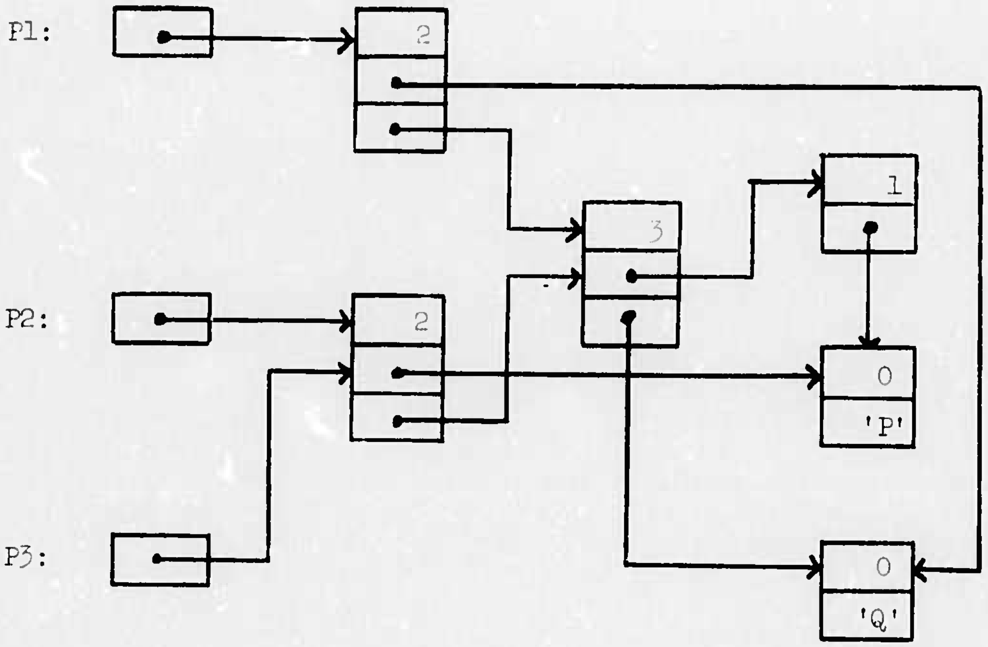
would be represented as:



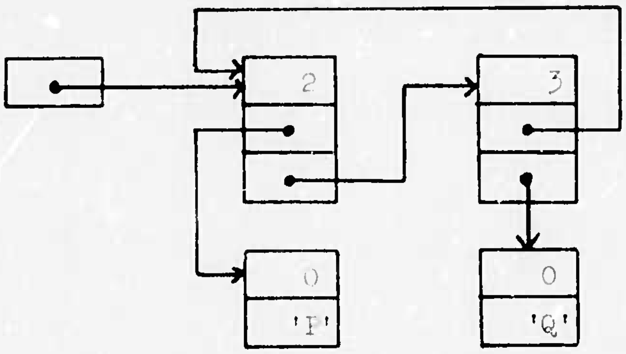
Of course, this example is untypically simple. A picture of a more realistic proposition would explain why the programmer may prefer not to think in terms of references.

On many machines it will be possible to pack the tag in with one of the components of the node, or pack two addresses in a single word, thereby saving a word of storage on that node. It can be seen that when nodes have more than two components it is possible to use less space than the standard LISP representation for the same information.

The call of a (non-constant) generator involves the dynamic acquisition of a few words of contiguous main storage, and planting in them the values of its simple parameters, and the addresses of its recursive parameters. The value returned by the generator is the address of the new node. There is no need to make a fresh copy of the recursive components, since it is quite permissible for two separate variables to "share" the same components, thus:



In this picture P1 has value Q & (¬ P ∨ Q) and P2 and P3 have the same value P & (¬ P ∨ Q) . However, this shared use of storage is entirely invisible to the programmer, who has no means of finding out whether it has occurred or not. This is because the prohibition on the selective updating of components of a structure prevents the programmer from changing a node on one tree, and testing to see whether the change has affected the other. The same restriction also prevents the establishment of cyclic structures, like:



Such a structure would appear to have the "infinite" value:

$$(P \,\&\, (P \,\&\, (P \,\&\, \dots \vee Q)) \vee Q) \quad ,$$

and this would fail to satisfy the axiom of finite generation. Thus the prohibition on selective updating seems to be a vital means of preserving the integrity of recursive data structures, as well as permitting a more economic "shared" representations.

The tree representation using addresses is not the only possible representation of recursive data structures. If the structure is to be held on backing store, it should be converted to a linear stream, replacing every address by the stream representing the tree to which it points. In this representation, the example $P \,\&\, (\neg P \vee Q)$ would appear:

2	0	'P'	3	1	0	'P'	0	'Q'

This, of course, will require copies to be taken of all shared branches, thereby usually occupying more space; but in general the elimination of addresses will compensate for this. Of course, on reinput of the structure, it would be advisable to reestablish as much sharing as possible; before acquiring a new node to accommodate given values, if a node already containing these values is already present in store, it should be used instead. Indeed, the reuse of existing storage in this way may be adopted as general policy, which can be effective in certain kinds of application -- for example, it makes test of equality very cheap; in any case, it is entirely invisible as far as the logic of the program is concerned. In a conventional non-associative store a hashing technique is recommended for finding a node with given contents; hence in LISP it is known as "the hashing cons" [9].

If sharing is used, it is no longer possible to reclaim all the storage allocated to a variable on exit from the block in which the variable was declared, since its components may also be components of the value of some variable global to that block. In order to reclaim storage when it runs out (and it soon will) it is necessary to use a scan-mark garbage collector invented by McCarthy for this purpose [10]. This will be more complicated than the standard LISP garbage collector, since it will have to deal in blocks of different size, and it will have to know the type of each node and the relative position of each address within it. In many applications, the size of the nodes do not vary too wildly; so the problem of fragmentation should not be significant. The cost per node of garbage collection should be no greater than in LISP, and if nodes are larger than two words, some saving in time may be possible.

The case expression can be compiled into highly efficient code. The value being considered may be loaded into an index register. The tag is then used to do an indexed jump leading to the portion of object code dealing with that particular case. There is no need to check the range of the index; it is logically impossible for it to be wrong! If there are more than two alternatives this could be more compact and efficient than a sequence of tests. The left hand side of the arrow generates no code at all. In compiling the right hand side, the formal parameters of this case can be accessed by means of a single reverse indexed instruction (Ross [11]) requiring a single store access. In accessing the third or subsequent component of a node this will be more compact and efficient than the LISP use of cadr , caddr , etc.

With reasonable cooperation from the programmer, this implementation would seem to offer a significant improvement on the efficiency of compiled LISP, perhaps even a factor of two in space x cost for suitable applications. But even more significant may be the fact that normal operations on numbers, characters, bits, etc., can be carried out with direct machine code instructions, without the preliminary run time type check which can be so cumbersome in compiled implementations of LISP. Thus the overall improvement might sometimes approach an order of magnitude.

4. Axioms

The axioms for a recursive data type are closely modelled on the corresponding informal definition of the type as given in the beginning of Section 2. Note that the fourth axiom is expressed quite informally: in its normal formalization it appears as a principle of "structural induction". Consider any predicate $\rho(q)$, which we wish to prove true of all propositions q, i.e., we wish to prove

$$\forall q\text{: proposition}.\,\rho(q) \quad .$$

The principle of structural induction states that this can be established by proving the theorem for all the ways in which a proposition q can be generated; and furthermore in these proofs ρ may be assumed true of all propositional components of q. This may be expressed in the proof rule:

$$\begin{array}{l}
\forall c\text{: letter}.\ \rho(\text{prop}(c)) \\
\forall p\text{: proposition}.\ \rho(p) \Rightarrow \rho(\text{neg}(p)) \\
\forall p,q\text{: proposition}.\ \rho(p)\ \&\ \rho(q) \Rightarrow \rho(\text{conj}(p,q))\ \&\ \rho(\text{disj}(p,q)) \\
\hline
\qquad \forall q\text{: proposition}.\ \rho(q)
\end{array}$$

The first three lines of this rule are the antecedents, and the last line is the conclusion of the deduction.

The fifth axiom, dealing with equality, is most easily formalized by giving axioms defining the meaning of the cases expression.

For propositions the axiom takes the form:

$$\begin{array}{ll}
5. & \underline{\text{cases}}\ \text{prop}(d)\ \underline{\text{of}}\ (\ldots|\text{prop}(c) \rightarrow e|\ldots) = e^{c}_{d} \\
& \&\ \underline{\text{cases}}\ \text{neg}(p)\ \underline{\text{of}}\ (\ldots|\text{neg}(q) \rightarrow e|\ldots) = e^{q}_{p} \\
& \&\ \underline{\text{cases}}\ \text{conj}(p,q)\ \underline{\text{of}}\ (\ldots|\text{conj}(r,s) \rightarrow e|\ldots) = e^{r,s}_{p,q} \\
& \&\ \underline{\text{cases}}\ \text{disj}(p,q)\ \underline{\text{of}}\ (\ldots|\text{disj}(r,s) \rightarrow e|\ldots) = e^{r,s}_{p,q}
\end{array}$$

where e^{x}_{y} means the expression formed from e by replacing all free occurrences of the variable x by the expression y (with appropriate modifications of bound variables when necessary).

The question now arises, are these axioms suffieiently powerful to prove everything we need to know about recursive data structures? Of course, this question is not precise enough to permit a definitive answer; but our confidence in the power of the axioms can be established by showing their close analogy with the Peano axioms for natural numbers, which have been found adequate for all practical purposes of arithmetic. For they too can be defined as recursive data structures:

```
        type NN = (zero, succ(NN));
```

and the cases notation permits the traditional method of defining recursive functions, for example:

```
        function plus (m,n: NN): NN;
           plus := cases n of (zero → m|succ(p) → suec(plus(m,p)));
```

The axioms for natural numbers defined as a recursive data structure are

(1) zero is an NN

(2) if n is an NN , so is succ(n)

(3)
$$\frac{\rho(\text{zero}) \qquad \forall n\text{: NN}.\rho(n) \;\Rightarrow\; \rho(\text{succ}(n))}{\forall n\text{: NN}.\ \rho(n)}$$

(4) cases zero of (zero → e | succ(n) → f) = e

& cases suce(m) of (zero → e | succ(n) → f) = f^{n}_{m}

Axioms (1) and (2) are the same as Peano's. Axiom (3) is the principle of mathematical induction. From Axiom (4) we can readily prove the remaining two Peano axioms:

1. succ(n) ≠ zero

Proof by contradiction: assume succ(n) = zero

∴ cases succ(n) of (zero → true | succ(n) → false)

= cases zero of (zero → true | succ(n) → false)

∴ false = true by axiom 4.

2. succ(m) = succ(n) ⇒ m = n

Proof: assume the antecedent.

$$\begin{aligned}\text{hence}\quad & \underline{\text{cases}}\ \mathrm{succ}(m)\ \underline{\text{of}}\ (\mathrm{zero} \rightarrow \mathrm{zero} \mid \mathrm{succ}(m) \rightarrow m)\\ & = \underline{\text{cases}}\ \mathrm{succ}(n)\ \underline{\text{of}}\ (\mathrm{zero} \rightarrow \mathrm{zero} \mid \mathrm{succ}(m) \rightarrow m)\\ \therefore\quad & m_m^m = m_n^m \quad , \quad \text{i.e.,} \quad m = n \quad .\end{aligned}$$

It is worthy of note that when none of the generators have parameters, the recursive data structure reduces to a PASCAL type definition by enumeration, and the axioms still remain valid. For example, the Boolean type may be defined:

$$\underline{\text{type}}\ \mathrm{Boolean} = (\mathrm{true} \mid \mathrm{false});$$

and the axioms are:

(1) true and false are Booleans,

(2)
$$\frac{\rho(\mathrm{true}) \qquad \rho(\mathrm{false})}{\forall b{:}\ \mathrm{Boolean}.\ \rho(b)}$$

(3)
$$\begin{aligned}&\underline{\text{cases}}\ \mathrm{true}\ \underline{\text{of}}\ (\mathrm{true} \rightarrow e \mid \mathrm{false} \rightarrow f) = e\\ &\underline{\text{cases}}\ \mathrm{false}\ \underline{\text{of}}\ (\mathrm{true} \rightarrow e \mid \mathrm{false} \rightarrow f) = f\end{aligned}$$

If desired, the notation " if B then e else f " may be regarded as an abbreviation for " cases B of (true → e | false → f) ".

5. Classes

Many interesting algebras are not word algebras -- for example, finite sets and finite mappings (sparse arrays). However, they can be represented as subsets of a word algebra, consisting of elements satisfying some additional property known as an invariant for that type. A type which is a subset of a word algebra will be called a class. In order to ensure that each newly generated value of the type will actually satisfy the invariant, the programmer must have the ability to specify

(1) The initial value of any declared variable of the class.

(2) The function(s) which are to be used to generate all other values of the class.

A programming language should ensure that the actual generators for the recursive class are never used outside the bodies of the function(s). In this way, by proving that these functions preserve the invariant (whenever their parameters satisfy it), it is possible to guarantee that all values ever generated will be within the desired subset. This idea was expounded in [12].

As an example, consider the representation of a set of integers. For this purpose we shall use a single-chained list of integers, which possesses the additional invariant property of being sorted. The operations required for a set are (say) insertion of a possibly new element, deletion of a possibly present element, and a test of membership of a possible element. A suggested form for the class declaration may be

```
(1)  class intset = (empty | list(intset, integer))
(2)     begin function insertion(s: intset, i: integer): intset;
           insertion := cases s of
                        (empty → list(empty,i)|
                        list(rest,j) → if i = j then s
                                   else if i > j then list(insertion(rest,i),j)
                                                else list(s,i));
           function deletion(s: intset, i: integer): intset;
              deletion := cases s of
                          (empty → empty|
                          list(rest,j) → if i = j then rest
                                     else if i > j then list(deletion(rest,i),j)
                                                  else s);
           function has(s: intset, i: integer): intset;
              has := cases s of
                     (empty → false|
                     list(rest,j) → if i = j then true
                               else if i > j then has(rest,i)
                                             else false);
(3)        intset := empty
     end intset;
```

Notes

(1) Introduces the class name intset , and declares that it will be a subset of the recursive type with generators empty and list. The scope of these generator names is confined to this class declaration.

(2) The body of the class declaration, as in [12], has the form of a block, in which are declared those procedures and functions which are to be used by the programmer on values of the class, namely the functions insertion and deletion and has .

(3) The body of the block specifies the initial value of all declared variables of the class. The name of the class itself is used for this purpose.

It is the intention that a class can be used in the same way as a type, for example:

declaration (including initialization to empty):	R,S: intset;

assignment:	R := insertion(S,37)
	S := R; R := deletion(R,56);

test:	if has(R,37) then ...

As suggested in [12], the criterion of correctness of a class can be expressed in terms of an *invariant* and an *abstraction function*.

The abstraction function which maps each list onto the set which it represents can be defined by recursion

$$\begin{aligned}\mathcal{A}(\ell\text{: intset}) =_{\text{df}}\ & \underline{\text{cases}}\ \ell\ \underline{\text{of}}\\ & (\text{empty} \rightarrow \text{null set}\,|\\ & \text{list}(\ell 1,\text{i}) \rightarrow \{\text{i}\} \cup \mathcal{A}(\ell 1));\end{aligned}$$

and the invariant can be expressed

$$\begin{aligned}\text{sorted}(\ell\text{: intset}) =_{\text{df}}\ & \underline{\text{cases}}\ \ell\ \underline{\text{of}}\\ & (\text{empty} \rightarrow \text{true}\,|\\ & \text{list}(\ell 1,\text{i}) \rightarrow \text{i} = \min(\mathcal{A}(\ell))).\end{aligned}$$

The correctness of the insertion function can now be formally expressed.

$$\text{sorted(s)}\{\text{body of insertion}\}\ \text{sorted(insertion)}\ \&\ \mathcal{A}(\text{insertion}) = \{\text{i}\} \cup \mathcal{A}(\text{s})$$

Since insertion is a recursive function, the proof of this will require assumption of the correctness of the recursive call, namely: " insertion(rest,i) ". This hypothesis may be expressed:

$$\begin{aligned}&[\text{sorted(rest)} \Rightarrow]\ \text{sorted(insertion(rest,i))}\ \&\\ &\qquad\&\ \mathcal{A}(\text{insertion(rest,i)}) = \{\text{i}\} \cup \mathcal{A}(\text{rest}) \ \ldots \text{hypothesis}\end{aligned}$$

in which the antecedent is true for all intsets, and may be omitted. Using the rule of assignment, and distributing function application through the cases, we obtain the following lemma:

```
sorted(s) ⇒ cases s of
        (empty → sorted(list(empty,i)) & $\alpha$(list(empty,i)) = {i} ∪ $\alpha$(s)          (1)
        list(rest,j) → if i = j then sorted(s) & $\alpha$(s) = {i} ∪ $\alpha$(s)             (2)
                else if i > j then sorted(list(insertion(rest,i),j))               (3)
                      & $\alpha$(list(insertion(rest,i),j)) = {i} ∪ $\alpha$(s)          (4)
                           else sorted(list(s,i))                                (5)
                               & $\alpha$(list(s,i)) = {i} ∪ $\alpha$(s)                 (6)
```

Each case can be readily proved from the definition of α and sorted ; no further inductions are required.

6. Memo Functions

In this section, we shall explore a particular case of selective updating of components of a recursive data structure, which enables the programmer to secure the advantages of the memo function advocated by Michie [13]. Consider the old example of differentiation of symbolic expressions. The simplest implementation is to define expressions as a type:

```
type expression = (constant(real) | variable(identifier)|
                   minus(expression)|
                   sum, product, quotient(expression, expression));
```

and define the derivative as follows:

```
function deriv(e: expression, t: identifier): expression;
   deriv := cases c of
            (constant(any) → constant(0)|
            variable(x) → if x = t then constant(1)
                                   else constant(0)|
            minus(u) → minus(deriv(u))|
            sum(u,v) → sum(deriv(u),deriv(v))|
            product(u,v) → sum(product(u,deriv(v)),product(v,deriv(u)))|
            quotient(x,y) →
               quotient(sum(deriv(u),product(minus(e),deriv(v))),v));
```

Using these declarations we may write:

```
        position, speed, acceleration: expression;
        position := quotient(constant(3),variable('t'));
        speed := deriv(position,variable('t'));
        acceleration := deriv(speed, variable('t'))  .
```

But this implementation can involve heavy penalties both in space and time:

(1) A large amount of space will be wasted in storing expressions of the form

$$e+0 \quad , \quad e\times 1 \quad , \quad e\times 0 \quad , \quad \text{etc.}$$

This may be mitigated by declaring expressions as a class, in which the generation of such redundant expressions is inhibited, by the use of programmed functions.

(2) If an expression is to be differentiated repeatedly with respect to the same variable, much time and space can be spent on re-evaluating the derivatives of the subexpressions; this time could be saved if the previously computed derivative were stored as a third component of each node representing a sum, a product or a quotient. The value of this component (known as a memo component) starts off as "unknown", but when the derivative of this subexpression is computed, it is stored here; and if the derivative is required again, the stored value is used instead of being recomputed.

For the sake of simplicity, in the following program we have assumed that all derivatives are taken with respect to the variable 't'; also, the functions perform only the most trivial of simplifications. In a serious symbolic manipulation program, all these functions would be much more complicated.

```
      class expression = (variable(identifier)|constant(real)|mi(expression)|
                         su,pr,qu(expression,expression,(unknown,known(expression))))
(1)    begin constant zero = constant(0), one = constant(1);
          function sum(left,right: expression): expression;
             sum := if left = zero then right else if right = zero then left
                                       else su(left,right,unknown);
          function minus(e: expression): expression;
             minus := cases e of (constant(x) → constant(-x)|
(2)                                 mi(f) → f | else mi(e));
          function product(left,right: expression): expression;
             product := if left = zero ∨ right = one then left
                   else if right = zero ∨ left = one then right
                            else pr(left,right,unknown);
          function quotient(left,right: expression): expression;
             quotient := if left = zero ∨ right = one then left
                                                   else qu(left,right,unknown);
```

```
              function dbydt(e: expression) → expression;
                 cases e of
                 (variable(id) → dbydt := if id = 't' then one else zero|
                 constant(any) → dbydt := zero|
                 mi(u) → dbydt := minus(dbydt(u))|
                 su(u,v,deriv) → cases deriv of
                                 (known(f) → f|
                                 unknown → {dbydt := sum(dbydt(u),dbydt(v));
(3)(4)                                      deriv := known(dbydt)})|
                 pr(u,v,deriv) → cases deriv of
                                 (known(f) → f|
                                 unknown → {dbydt := sum(product(u,dbydt(v)),
                                                     product(v,dbydt(u)));
                                           deriv := known(dbydt)})|
                 qu(u,v,deriv) → cases deriv of
                                 (known(f) → f|
                                 unknown → {dbydt := quotient(sum(dbydt(u),
                                                 minus(product(e,dbydt(v)))),v);
                                                 deriv := known(dbydt)}));
              expression := zero
          end expression;
```

Notes

(1) The PASCAL constant declaration can here be used to save space and time and trouble.

(2) It seems a convenience to write else to stand for all the cases not explicitly mentioned.

(3) It is also convenient to use the name of a function as a variable inside its body (except, of course, when it has actual parameters).

(4) { } are used for begin end.

The correctness of this class obviously depends on the preservation of the invariant that if e has the form su , pr , or qu , then its memo component either contains the value unknown or known(dbydt(e)) ; or, more formally:

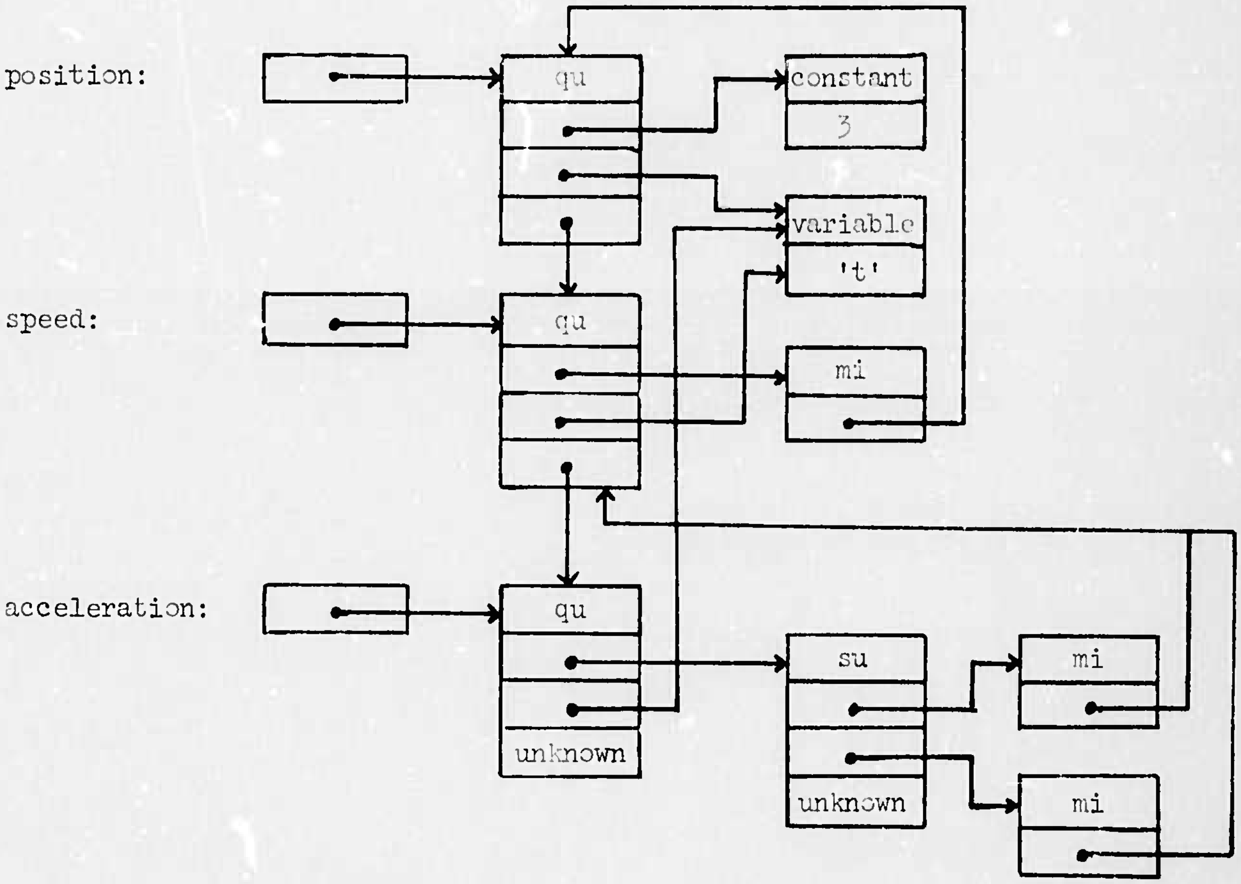
$$\begin{aligned}\forall e,u,v,d:\ \text{expression}.\ & e = su(u,v,known(d)) \vee e = pr(u,v,known(d)) \\ & \vee e = qu(u,v,known(d)) \Rightarrow d := dbydt(e))\quad .\end{aligned}$$

Furthermore the abstraction function for the class must not mention the memo component. It is this that makes the existence of the third component logically invisible to the user of the class, although one hopes that he notices the gain in efficiency.

It is noteworthy that the use of selective updating immediately permits establishment of cyclic structures, but because of its logical invisibility, this does not seem to matter. For example, after a series of assignments like those shown on page 21,

```
position, speed, acceleration: expression;
   position := quotient(constant(3),variable('t'));
   speed := dbydt(position);
   acceleration := dbydt(speed);
```

A diagram of the stored structures will be:



(<u>Note</u>: position = 3/t

$\text{speed} = -\text{position}/t = -3/t^2$

$\text{acceleration} = \dfrac{(-\text{speed}) + (-\text{speed})}{t} = \dfrac{3/t^2 + 3/t^2}{t} = \dfrac{6}{t^3}$.)

The use of memo components does not invalidate the sharing of subtrees, again because of the invisibility of the updating. Indeed, its main benefits are directly due to the preservation of sharing, and can be increased by increasing the amount of sharing. If the memo function method is widely used, it becomes very attractive to choose the "hashing" technique of storage allocation. However, in the use of this technique, it would be desirable to ignore the contents of the memo component, so that if a newly generated expression was identical to one in which the memo component was already known, they would still be correctly identified, and the derivative of the newly generated expression would be available "for free". For this reason, it would seem to be a good idea for a programming language to insist that a programmer single out a memo component by a special form of declaration, say by prefixing it by the word memo . A similar method has been used successfully in some large theorem proving systems [14].

The language feature defined here places on the programmer the responsibility for correct maintenance of a memo component; and it helps him in this only by supplying an appropriate proof method. This has the advantage that the programmer can readily control the nature and amount of information to be memorized. For example, if partial derivatives are required with respect to exactly three variables, three memo components can be declared. If the identity of the controlled variable is not known in advance, it can also be stored in a memo component, so that repeated differentiation with respect to the same variable will always be efficient, although when a different variable is used, the memory is overwritten. Or the programmer can maintain a small list of such variable/value pairs, choosing to "forget" certain of them when the list gets too long. Finally, he can choose which nodes will have memo components and which will not. This gives the programmer much better control of efficiency in time and storage than the automatic technique suggested in [13], although at a cost of requiring correct programming. Since efficiency is the sole objective of the memo technique, perhaps this is not too high a price.

The implementation of this memo technique perhaps constitutes one of the better disciplined uses of the controversial LISP functions RPLACA and RPLACD.

7. Non-shared Representations

In the previous sections we have given examples for which an implementation using shared substructures would give significant savings in storage space and time. However, the use of storage sharing has some significant penalties:

(1) when updating any component of any node of a structure, a new copy must be made of that node and all nodes through which it was accessed;

(2) storage which goes out of use, either because of updating or because of block exit, cannot be immediately reclaimed for other uses;

(3) the programmer tends to lose control of the efficiency of use of one of his most precious assets, main storage;

(4) the programmer has no control over addressing vagrancy, which is necessary for successful use of paging systems or backing stores;

(5) the time spent in scan-mark garbage collection can be the heaviest single cost in the execution of an efficiently compiled program.

These disadvantages will be particularly acute in cases where little advantage can be taken of sharing.

Consider for example a program operating on intsets (as defined in Section 5) which only ever needs one such set; or if it needs several, it only ever updates the sets by assignments of the form

```
S1 := insertion(S1,57);
S2 := deletion(S2,93);
```

and never performs a "cross-assignment" of the form:

```
S1 := S2;
S2 := insertion(S1,57).
```

In such a program, the two sets would never in practice come to share any subcomponent. Even if the program did make an occasional cross-assignment, the sharing patterns would be rapidly dissipated by subsequent updating of either set. So in this program a non-shared representation would be much better.

If it is known that there is no sharing, the programmer must be encouraged to use selective updating of components of his structure by means of procedures operating upon the structure, rather than functions producing potentially large structured values. As suggested in [12], we shall declare *procedures* local to the class:

```
        procedure insert(i: integer);
        procedure delete(i: integer);
```

These procedures are regarded as being "components" of every variable of the class, and can be invoked by naming the variable followed by the procedure call (separated by a dot):

```
        S1.insert(57);
        S2.delete(93);
```

which are intended to be equivalent to the updating assignments

```
        S1 := insertion(S1,57);
        S2 := deletion(S2,93).
```

The writer of these procedures sometimes wishes to refer to the yet unknown variable to which it is being applied. For this purpose, we will use the name of the class itself. In some circumstances, it is necessary to define a completely new value of this variable by means of a generator. For this purpose, I suggest a facility used in many languages to specify the result of a function, namely:

```
        result e;
```

which has the effect of assigning e as the result of the procedure, and immediately exiting from the procedure. The code for the updating version of intset is shown in Table 1.

When the result of a procedure is given by *result* , part or all of the store used by the value of the variable being updated can often be immediately reclaimed -- a technique which has been called "compile time garbage collection" [15]. An example of this is marked (2).

One consequence of a non-shared representation is that whenever a structure is used as an argument to a generator, a complete copy of that structure must be made. An exception to this is in the case of a *single* occurrence of the class name within an expression which is being used as

```
class intset = (empty|list(intset,integer))

begin procedure insert(i: integer);
   cases intset of (empty → result list(empty,i)|
                   list(rest,j) → if i > j then rest. insert(i)
                            else if i < j then result list(intset,i));    (1)
      procedure delete(i: integer);
   cases intset of (empty → do nothing|
                   list(rest,j) → if i = j then result rest                (2)
                            else if i > j then rest. delete(i));
      function has(i: integer): Boolean;
   cases intset of (empty → result false|
                   list(rest,j) → if i = j then result true
                            else if i > j then result rest. has(i));
      intset := empty
end;
```

Table 1.

a result. In this case, the same address can be used instead of the address of a fresh copy; and of course, the storage occupied would not be reclaimed. An example of this is marked (1).

After these two optimizations have been made, the outstanding causes of inefficiency are the recursive calls, with their associated overhead of stack manipulation and parameter passing. Since these will occur as the last statement of the procedure body, an obvious optimization would be to replace them by a jump back to the beginning of the procedure body, having made appropriate adjustment for the left hand parameter of the procedure.

After these optimizations have been made, the resulting program may in certain applications be several orders of magnitude more efficient than the purely functional class described in Section 5.

It is unfortunate that the language feature described here relies so heavily on optimization to secure highest efficiency. The great danger of optimization is that a small change to a program (e.g. insertion of n := n+1 after, instead of before, a recursive call) will give rise to an unpredictable and unacceptable loss of efficiency. A second danger is that it can make an implementation, large, slow, unreliable, and late. Finally, it has the unfortunate effect of removing from the programmer the feeling of responsibility and control over efficiency, which was the main reason for introducing selective updating anyway!

Consequently, it may be desirable to introduce into a language some special notations for expressing the three special cases which are susceptible to optimization.

Conclusion

This paper has described a number of old programming techniques and new notations to express them. The objective has been to isolate a number of useful and efficient simple cases of the use of references, which are susceptible of relatively simple proof techniques, and give notations and syntactic conventions which guarantee their validity. In all cases it has proved possible to achieve this without introducing the concept of a reference into the algorithms. Of course there must remain a number of applications, for example when dealing with relational structures, when the explicit use of references seems unavoidable or even desirable; and for this purpose, something like the record class of ALGOL W still seems to be the best solution.

However, if a general-purpose programming language contains a record class concept, this can certainly be used to program all the representations described in this paper. So the question arises, is it worth while to incorporate these notations and facilities into such a language? The answer to this probably depends on the intended application of the language. In a special purpose language for symbol manipulation it would be interesting to try out some of these ideas; but in a general language for implementing software (e.g. operating systems), the reliance on general garbage collection seems quite inappropriate.

Even if these ideas are not embodied in a programming language, I hope they will be found to be useful as an aid to reliable program design and documentation. An algorithm can be designed using the suggested facilities, and can perhaps even proved using the suggested proof techniques; the programmer can then manually translate his abstract program into some lower level language with explicit pointers, using the suggested implementation techniques. This is, of course, the general method recommended in structured programming [16].

References

[1] ed. van Wijngaarden. "Report on the Algorithmic Language ALGOL 68," *Num. Math.* 14 (1969), 79-218.

[2] "PL/I Language Specifications," IBM Order Number GY33-6003-2.

[3] ed. P. Naur. "Report on the Algorithmic Language ALGOL 60," *Num. Math.* (1960), 106-136.

[4] N. Wirth and C. A. R. Hoare. "A Contribution to the Development of ALGOL," *Comm. ACM* 9, 6 (June 1966).

[5] N. Wirth. "The Programming Language PASCAL," *Acta Informatica* 1, 1 (1971), 35-63.

[6] D. E. Knuth. "A Review of Structured Programming," STAN-CS-73-371 (June 1973).

[7] J. McCarthy. "A Basis for a Mathematical Theory of Computation," in *Computer Programming and Formal Systems*, (ed. Braffort and Hirschberg), North Holland (1963).

[8] F. V. McBride, D. J. T. Morrison, R. M. Pengelly. "A Symbol Manipulation System," *Machine Intelligence 5*. Edinburgh University Press (1970).

[9] LISP folklore.

[10] J. McCarthy. "Recursive Functions of Symbolic Expressions and their Computation by Machine," *Comm. ACM* 3, 4 (April 1960), 184-195.

[11] D. T. Ross. "A Generalized Technique for Symbol Manipulation and Numerical Calculation," *Comm. ACM* (March 1961).

[12] C. A. R. Hoare. "Proof of Correctness of Data Representations," *Acta Informatica* 1 (1972), 271-281.

[13] D. Michie. "Memo Functions: a Language Feature with Rote Learning Properties," DMIP Memorandum MIP-R-29 (November 1967).

[14] R. Waldinger and K. N. Levitt. "Reasoning about Programs," Proceedings of ACM Sigact/Sigplan Symposium on Principles of Programming Language Design, Boston, 1973.

[15] J. Darlington and R. M. Burstall. "A System which Automatically Improves Programs," Proceedings of Third International Conference on Artificial Intelligence, Stanford, 1973.

[16] E. W. Dijkstra. "Notes on Structured Programming," in Structured Programming, Academic Press (1972).